More Advice on Proving a Compiler Correct:

Improve a Correct Compiler

E. Meijer

UU-CS-1994-05
January 1994

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

More Advice on Proving a Compiler Correct:

Improve a Correct Compiler

E. Meijer

Technical Report UU-CS-1994-05
January 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

More Advice on Proving a Compiler Correct: Improve a Correct Compiler

Erik Meijer
University of Utrecht
Department of Computer Science
POBox 80.089
NL-3508 TB Utrecht
The Netherlands
email: erik@cs.ruu.nl

Abstract

Pragmatically speaking, a denotational semantics for a programming language L is just an interpreter $\mathcal{M}[\![\_]\!] \in \mathbb{L} \to M$ for L written in a functional meta language. When such an interpreter is written in a catamorphic, or compositional, style, it is possible to factor it automatically into a compiler from the original source language L into a new target language T and a residual interpreter for T. Moreover given a homomorphism $e \in M \to \mathbb{U}$ from the semantic domain M into another domain U, the fusion law yields a new catamorphic interpreter $e \circ \mathcal{M}[\![\_]\!] \in \mathbb{L} \to \mathbb{U}$. This new interpreter can in turn be factored into a compiler and a residual interpreter.

The game we want to play then is to transform an initial abstract interpreter that only tells what the source language means into a final concrete interpreter that tells how program denotations are computed, via a chain of efficiency improving homomorphisms, knowing that along the way each intermediate interpreter can be turned into a compiler. Hence we have reduced the compiler correctness problem into a functional programming problem.

As an example a compiler is derived for a simple imperative language with first order procedures that generates conventional three address code.

1 Introduction

One of the original objectives of denotational semantics is to give a precise description of programming languages that can serve as a standard against which implementations can be verified, or preferably, from which correct compilers can be derived. We want to use a *calculational* approach to derive correct and efficient implementations for programming languages from their denotational descriptions. Besides the correctness aspect, a transformational approach can help to understand the relationships that may or may not exist between various implementations, or even suggest alternative methods.

In contrast to the consensus about formal definition of syntax by means of some variant of context free grammars, such as BNF, there is no agreement about a concise, readable and easily manipulatable notation for specifying the semantics of programming languages. Following the adagium "the essence of denotational semantics is the translation of conventional programs into equivalent functional programs" [17], we propose as semantic meta-language a variant of the Squiggol style of programming [18, 2] based on CPOs [20] combined with the pragmatic aspects of Action Semantics [23]. This blend covers traditional techniques such as Initial Algebra Semantics [9] and Attribute (or Affix) Grammars [6, 13, 14].

ISSN: 0924–3275

The idea of defining a language by giving an interpreter for it and transforming that interpreter into a compiler also lies at the very hart of partial evaluation. Not surprisingly our work bears some resemblance to partial evaluation. Many of the tricks that we use to improve our derivations also arise in tuning interpreters for effective partial evaluation. The main difference is that we do not strive for fully mechanical generation of compilers from interpreters using one of the Futurama projections [10], instead we rely on a powerful factorization theorem that gives a simple recipe for transforming an interpreter into a compiler. Having no automation in mind allows us to bypass a formal type system to distinguish between static and dynamic values. Instead we make the distinction informally, but very carefully.

This paper is a condensed version of the author's PhD thesis [19]. Besides the compiler for the imperative language described in this paper, the thesis derives implementations of a simple functional and a simple logic programming language.

2 The Compiler Correctness Problem

Many people [12, 21, 24, 3, 30, 22, 5, 4] have suggested the use of algebraic means to tackle the compiler correctness problem. Given a source language L, a target language T, their respective semantics $\mathsf{m} \in \mathsf{L} \to \mathsf{M}$, $\mathsf{a} \in \mathsf{T} \to \mathsf{U}$ and a compiler c from L to T, one seeks an embedding e of the source semantics into the target semantics such that the following diagram commutes.

$$\begin{tikzcd}
\mathsf{L} \arrow[r, "\mathsf{c}"] \arrow[d, "\mathsf{m}"'] & \mathsf{T} \arrow[d, "\mathsf{a}"] \\
\mathsf{M} \arrow[r, "\mathsf{e}"'] & \mathsf{U}
\end{tikzcd}$$

By enforcing an algebraic structure on the different domains and defining the respective functions as homomorphisms, initiality of L ensures commutativity. If L is initial, homomorphisms from L to any other algebra are unique; homomorphisms $\mathsf{a} \circ \mathsf{c}$ and $\mathsf{e} \circ \mathsf{m}$ both go from L to U and hence they are equal. A sufficient condition to prevent trivial solutions which result from taking U as a final algebra, is to require e to be injective, i.e. U must be a true implementation of M. The above classic approach has not been concerned with calculational issues. It only provides a framework for proving the correctness of a given compiler. We want to derive a new compiler by calculation not to prove a given one correct. Where in the classical approach one looks for the embedding e in the compiler correctness equation $\mathsf{a} \circ \mathsf{c} = \mathsf{e} \circ \mathsf{m}$, we want to construct an efficient compiler c knowing only the semantics m. Excellent surveys of other work in the area of semantics directed compiler generation are given by Tofte [29] and Schmidt [27, Chapter 10]. The proceedings of a workshop in 1980 [11] still provides many interesting references as well.

3 Algebras

In order to try and find that more constructive method we look a little more closely into the algebraic framework. Don't be afraid of all the squiggles and formulea that will be introduced, after we have developed enough theory we switch back to more conventional functional programming style. It is just that the Squiggol notation lends itself better for proving generic high level theorems about any data type. Otherwise we end up with a catalogue of very similar rules, one for each recursive data type of interest, which makes it hard to see the wood for the trees. Actual calculations on the other hand are best done by instantiating the appropriate laws for a specific type and using a traditional notation.

An F-algebra is a pair $(\mathsf{A}, \varphi \in \mathsf{F}\,\mathsf{A} \to \mathsf{A})$ consisting of the *carrier* set A and a strict operation φ of *signature* F, for some suitable functor F. A *homomorphism* h between F-algebras (A, φ) and (B, ψ)

is a strict structure preserving map $\mathsf{h} \in \mathsf{A} \to \mathsf{B}$ that commutes the operation φ with ψ, formally, $\mathsf{h} \circ \varphi = \psi \circ \mathsf{F}\,\mathsf{h}$, or as a diagram

$$\begin{tikzcd}
\mathsf{F}\,\mathsf{A} \arrow[r, "\varphi"] \arrow[d, "\mathsf{F}\,\mathsf{h}"'] & \mathsf{A} \arrow[d, "\mathsf{h}"] \\
\mathsf{F}\,\mathsf{B} \arrow[r, "\psi"'] & \mathsf{B}
\end{tikzcd}$$

An invariant [8] $\mu\mathsf{F}$ of functor F is an F-algebra $\mu\mathsf{F} = (\mathsf{L}, \mathsf{in})$ where $\mathsf{in} \in \mathsf{F}\,\mathsf{L} \to \mathsf{L}$ is a bijection with inverse $\mathsf{out} \in \mathsf{L} \to \mathsf{F}\,\mathsf{L}$. Since out is just another function, we can use it to define a homomorphism (called *catamorphism*) from invariant $(\mathsf{L}, \mathsf{in})$ to any other algebra (A, φ) using the so-called 'banana'-brackets:

$$\begin{array}{rcll}
(\!|\mathsf{in} := \_|\!) & \in & (\mathsf{F}\,\mathsf{A} \to \mathsf{A}) \to (\mathsf{L} \to \mathsf{A}) & \text{(CataType)} \\
(\!|\mathsf{in} := \varphi|\!) & = & \mu(\lambda \mathsf{f}.\varphi \circ \mathsf{F}\,\mathsf{f} \circ \mathsf{out}) & \text{(Self)}
\end{array}$$

The *free theorem* or *parametricity condition* [31] for (CataType) is the *fusion law*, that tells us the composition of a homomorphism with a catamorphism is again a catamorphism (it is not true in general that catamorphisms are closed under composition).

$$(\!|\mathsf{in} := \psi|\!) = \mathsf{f} \circ (\!|\mathsf{in} := \varphi|\!) \;\Leftarrow\; \psi \circ \mathsf{F}\,\mathsf{f} = \mathsf{f} \circ \varphi \;\wedge\; \mathsf{f} \text{ strict} \qquad \text{(Fusion)}$$

If furthermore we assume that $(\mathsf{L}, \mathsf{in})$ is a *minimal* invariant, that is the copy function $(\!|\mathsf{in} := \mathsf{in}|\!)$ equals the identity id, we can show from (Self), (Fusion) and minimality that $(\!|\mathsf{in} := \varphi|\!)$ is the unique fixed point of $\lambda \mathsf{f}.\varphi \circ \mathsf{F}\,\mathsf{f} \circ \mathsf{out}$, i.e. that $(\mathsf{L}, \mathsf{in})$ is the initial F-algebra. In this paper we don't need minimal invariants but rely solely on the fusion law. The existence of (minimal) invariants is guaranteed by Scott's inverse limit construction [26].

In the sequel we will need to consider catamorphisms of higher type, for example $(\!|\mathsf{in} := \_|\!) \in (\mathsf{F}(\mathsf{A} \to \mathsf{B}) \to (\mathsf{A} \to \mathsf{B})) \to \mathsf{L} \to (\mathsf{A} \to \mathsf{B})$. The parametricity condition for this type is for strict f

$$\frac{\mathsf{F}(\circ\, \mathsf{g})\; \mathsf{a} = \mathsf{F}(\mathsf{f}\, \circ)\; \mathsf{b} \Rightarrow \varphi\; \mathsf{a} \circ \mathsf{g} = \mathsf{f} \circ \psi\; \mathsf{b}}{(\!|\mathsf{in} := \varphi|\!)\; \mathsf{as} \circ \mathsf{g} = \mathsf{f} \circ (\!|\mathsf{in} := \psi|\!)\; \mathsf{as}}$$

When instantiated for a particular F this usually expands into a monstruous formula. Its interpretation is that f distributes over $(\!|\mathsf{in} := \psi|\!)\;\mathsf{as}$ if f distributes over $\psi\;\mathsf{b}$ under the inductive hypothesis that f distributes over b.

Elements of an algebra can be defined recursively, and in that case the following leapfrog law turns out to be handy to compute the result of applying a catamorphism to such an infinite value.

$$(\!|\mathsf{in} := \varphi|\!)\;(\mu\,\mathsf{f}) = \mu\,\mathsf{g} \;\Leftarrow\; (\!|\mathsf{in} := \varphi|\!) \circ \mathsf{f} = \mathsf{g} \circ (\!|\mathsf{in} := \varphi|\!) \qquad \text{(CataMu)}$$

Property (CataMu) can be proved easily by fixed point induction.

3.1 Functional programming

The notion of (minimal) invariant coincides with algebraic data types as found in modern functional languages. For example the type of lists over A defined as

$$\mathsf{List}\;\mathsf{A} \;::=\; [\,] \mid \mathsf{A} : \mathsf{List}\;\mathsf{A}$$

corresponds to the minimal invariant $(\mathsf{List}\;\mathsf{A}, [\,] \triangledown (:))$ of the functor $\mathsf{F}\,\mathsf{X} = 1 + \mathsf{A} \times \mathsf{X}$. Catamorphisms $(\!|[\,] := \mathsf{c}, (:) := (\oplus)|\!)$ are generalized fold-operators that recursively replace the constructors $[\,]$ and: by respectively the operations c and $\oplus$.

$$\begin{array}{rcl}
\mathsf{fold}\;\mathsf{c}\;(\oplus)\;[\,] & = & \mathsf{c} \\
\mathsf{fold}\;\mathsf{c}\;(\oplus)\;(\mathsf{a} : \mathsf{as}) & = & \mathsf{a} \oplus \mathsf{fold}\;\mathsf{c}\;(\oplus)\;\mathsf{as}
\end{array}$$

Instantiating the first order fusion law, gives a familiar theorem for fold.

$$\frac{\begin{array}{l}\bullet\ \mathsf{f}\ \text{strict}\\ \bullet\ \mathsf{f}\ \mathsf{c} = \mathsf{d}\\ \bullet\ \mathsf{f}\ (\mathsf{a} \oplus \mathsf{as}) = \mathsf{a} \otimes (\mathsf{f}\ \mathsf{as})\end{array}}{\mathsf{f} \circ \mathsf{fold}\ \mathsf{c}\ (\oplus) = \mathsf{fold}\ \mathsf{d}\ (\otimes)}$$

For lists, the second order fusion law is still managable compared to that for the abstract syntax of our programming languages (see section 8.2).

$$\frac{\begin{array}{l}\bullet\ \mathsf{f}\ \text{strict}\\ \bullet\ \mathsf{f}\ (\mathsf{a}\ \mathsf{x}) = \mathsf{d}\ (\mathsf{b}\ \mathsf{x})\\ \bullet\ \mathsf{f}\ (\mathsf{c} \oplus \mathsf{d}\ \mathsf{y}) = \mathsf{c} \otimes \mathsf{e}\ (\mathsf{g}\ \mathsf{y}) \Leftarrow \mathsf{f}\ (\mathsf{c}\ \mathsf{z}) = \mathsf{e}\ (\mathsf{g}\ \mathsf{z})\end{array}}{\mathsf{f}\ (\mathsf{fold}\ \mathsf{a}\ (\oplus)\ \mathsf{as}\ \mathsf{k}) = \mathsf{fold}\ \mathsf{b}\ (\otimes)\ (\mathsf{g}\ \mathsf{k})}$$

We want to stress that any algebraic data type, even mutually recursive ones, corresponds to the minimal invariant of a functor that describes the signature of the type. Catamorphisms on these types are fold operators that recursively replace constructors by other functions of similar type and the fusion law is the free theorem of that fold operator. Minimal invariants and functors are just a convenient way of talking about algebraic types in general that circumvents sloppy notation like

$$\begin{array}{rcl}\mathsf{T} & ::= & \ldots \mid \mathsf{C}_i\ \mathsf{T}_1 \ldots \mathsf{T}_m \mid \ldots\\ \mathsf{fold}_\mathsf{T}\ \ldots \mathsf{c}_i \ldots\ \mathsf{t} & = & \mathsf{case}\ \mathsf{t}\ \mathsf{of}\ \ldots \mathsf{C}_i\ \mathsf{t}_1 \ldots \mathsf{t}_m \rightarrow \mathsf{c}_i\ (\mathsf{fold}_{\mathsf{T}_1}\ \mathsf{t}_1) \ldots (\mathsf{fold}_{\mathsf{T}_m}\ \mathsf{t}_m)\end{array}$$

3.2 Compositional semantics and catamorphisms

The abstract syntax of a programming language L can be defined as an initial F-algebra $(\mathsf{L}, \mathit{in})$ where in is the set of constructors of the abstract syntax. Therefore we can give an interpreter $\mathit{m} \in \mathsf{L} \rightarrow \mathsf{M}$ for L programs — a compositional denotational semantics for L — as the catamorphism $(\!|\mathit{in} := \varphi|\!)$ by imposing an F-algebraic structure (M, φ) on the semantic domain M. The important thing of a denotational semantics is to give the meaning of a construct $(\!|\mathit{in} := \varphi|\!) \circ \mathit{in}$ solely in terms of the meaning of the parts $\varphi \circ \mathsf{F}(\!|\mathit{in} := \varphi|\!)$. This is captured precisely by using a catamorphism for the semantic function.

In a traditional denotational description casting the semantic domain into the right form is achieved by encoding φ using λ-abstraction and application. Such language descriptions have rather poor pragmatic qualities. It is hard to identify essential semantic concepts of the language being described, and the (automatic) generation of compilers is virtually impossible. *Action Semantics* as developed by Mosses and Watt [33, 23] is an attempt to improve the readability and modularity of formal descriptions of programming languages. The semantic domain M is cast into a G-algebra (M, α) where the set of *actions* α corresponds to the run-time concepts of the programming language in question. For an imperative language an action algebra includes primitive actions such as assignment and action combinators such as sequencing and conditionals. The essence of writing an interpreter for L is transforming this fixed action algebra $\alpha \in \mathsf{G}\,\mathsf{M} \rightarrow \mathsf{M}$, into a compile-time algebra $\mathsf{T}\alpha \in \mathsf{F}\,\mathsf{M} \rightarrow \mathsf{M}$ of the same signature as the abstract syntax by means of a polymorphic *transformer* [7]

$$\mathsf{T} \in (\mathsf{G}\,\mathsf{A} \rightarrow \mathsf{A}) \rightarrow (\mathsf{F}\,\mathsf{A} \rightarrow \mathsf{A}) \qquad \text{(Transformer)}$$

The key property of transformers that we will use is that given transformer T, any interpreter $(\!|\mathit{in} := \mathsf{T}\alpha|\!)$ derived from action G-algebra (M, α) can be factored constructively into a compiler $(\!|\mathit{in} := \mathsf{T}\,\mathsf{IN}|\!)$ from source $(\mathsf{L}, \mathit{in})$ to F-algebra $(\mathsf{S}, \mathsf{T}\,\mathsf{IN})$ and a residual interpreter $(\!|\mathsf{IN} := \alpha|\!)$ from the initial G-algebra $(\mathsf{S}, \mathsf{IN})$ to the original domain (M, α).

$$\begin{tikzcd}
\mathsf{L} \arrow[rr, "\llparenthesis \mathsf{in} := \mathsf{T\ IN} \rrparenthesis"] \arrow[dr, "\llparenthesis \mathsf{in} := \mathsf{T}\alpha \rrparenthesis"'] & & \mathsf{S} \arrow[dl, "\llparenthesis \mathsf{IN} := \alpha \rrparenthesis"] \\
& \mathsf{M} &
\end{tikzcd}$$

In order to prove the factorization law, we use the parametricity condition for transformer T from G-algebras to F-algebras:

$$\mathsf{f} \circ \beta = \alpha \circ \mathsf{G\,f} \;\Rightarrow\; \mathsf{f} \circ \mathsf{T}\beta = \mathsf{T}\alpha \circ \mathsf{F\,f} \qquad \text{(Transformer)}$$

It tells us that homomorphisms on G-algebras are homomorphisms on the transformed F-algebras too. This is exactly what you would expect intuitively of a *polymorphic* transformation from G- to F-algebras. Correctness of the factorization theorem follows immediately from (Transformer) and (Fusion).

$$\begin{array}{ll}
& \llparenthesis \mathsf{in} := \mathsf{T}\alpha \rrparenthesis = \llparenthesis \mathsf{IN} := \alpha \rrparenthesis \circ \llparenthesis \mathsf{in} := \mathsf{T\ IN} \rrparenthesis \\
\Leftarrow & \quad \text{(Fusion)} \\
& \llparenthesis \mathsf{IN} := \alpha \rrparenthesis \circ \mathsf{T\ IN} = \mathsf{T}\alpha \circ \mathsf{F} \llparenthesis \mathsf{IN} := \alpha \rrparenthesis \\
\Leftarrow & \quad \text{(Transformer)} \\
& \llparenthesis \mathsf{IN} := \alpha \rrparenthesis \circ \mathsf{IN} = \alpha \circ \mathsf{G} \llparenthesis \mathsf{IN} := \alpha \rrparenthesis \\
\equiv & \quad \text{(Self)} \\
& \mathsf{true}
\end{array}$$

Factoring an interpreter into a compiler and a residual interpreter is a disciplined form of partial evaluation. The work done at compile-time is specializing a value in F-algebra $(\mathsf{S}, \mathsf{T\ IN})$ to a program in the G-algebra $(\mathsf{S}, \mathsf{IN})$. In ordinary partial evaluation IN (and α) are the unknowns in the factorization process and it is not known a priori what the signature of residual programs will be.

Usually the compiler generated as described above will not produce very efficient code because the difference between the source algebra and the target algebra of the transformer is too small. The remedy is to *improve* that already correct compiler. Improving a compiler derived from interpreter $\mathsf{m} = \llparenthesis \mathsf{in} := \mathsf{T}\alpha \rrparenthesis$ means finding an injective homomorphism e from $(\mathsf{M}, \mathsf{T}\alpha)$ to some new F-algebra $(\mathsf{U}, \mathsf{R}\beta)$. The fusion law then yields a more efficient interpreter from L to U provided we can prove that e is a homomorphism.

$$\begin{tikzcd}
& (\mathsf{L}, \mathsf{in}) \arrow[dl, "\llparenthesis \mathsf{in} := \mathsf{T}\alpha \rrparenthesis"'] \arrow[dr, "\llparenthesis \mathsf{in} := \mathsf{R}\beta \rrparenthesis"] & \\
(\mathsf{M}, \mathsf{T}\alpha) \arrow[rr, "\mathsf{e}"'] & & (\mathsf{U}, \mathsf{R}\beta)
\end{tikzcd}$$

The new transformer R, from the new action algebra $(\mathsf{U}, \beta \in \mathsf{H\,U} \to \mathsf{U})$ say, to the compile-time F-algebra, usually arises as side-effect of establishing the premise $\mathsf{e} \circ \mathsf{T}\alpha = \mathsf{R}\beta \circ \mathsf{F\,e}$ of the fusion law, i.e. of the proof that e is a homomorphism.

Injectivity of e ensures that the improved interpreter is equivalent to the previous one

$$\begin{array}{ll}
& \llparenthesis \mathsf{in} := \mathsf{T}\alpha \rrparenthesis \\
= & \quad \text{e injective} \\
& \mathsf{e}^{-1} \circ \mathsf{e} \circ \llparenthesis \mathsf{in} := \mathsf{T}\alpha \rrparenthesis \\
= & \quad \text{(Fusion)} \\
& \mathsf{e}^{-1} \circ \llparenthesis \mathsf{in} := \mathsf{R}\beta \rrparenthesis
\end{array}$$

hence $(\!|\mathsf{in} := \mathsf{T}\alpha|\!)$ can be replaced safely by $e^{-1} \circ (\!|\mathsf{in} := \mathsf{R}\beta|\!)$.

Based on the new interpreter $n = (\!|\mathsf{in} := \mathsf{R}\beta|\!)$ we can generate an improved compiler $c \in L \to T$ that solves the original compiler correctness diagram.

$$\begin{tikzcd}
& L \arrow[ddl, "m"'] \arrow[ddr, "n"] & & & L \arrow[rr, "c"] \arrow[ddr, "n"'] & & T \arrow[ddl, "a"] & & L \arrow[r, "c"] \arrow[dd, "m"'] & T \arrow[dd, "a"] \\
& & & + & & & & = & & \\
M \arrow[rr, "e"'] & & U & & & U & & & M \arrow[r, "e"'] & U
\end{tikzcd}$$

In many cases, for example when U is a function type $V \to W$, the above method does not yield a satisfying underlying action algebra. In that case we look for an $\mathsf{R}\beta$ such that $e \circ (\!|\mathsf{T}\alpha|\!)\, l = (\!|\mathsf{R}\beta|\!)\, l \circ e'$ and use the higher order fusion law to get a new interpreter, this time $(e' \circ) \circ (\!|\mathsf{in} := \mathsf{R}\beta|\!)$. This interpreter can subsequently be split into a compiler and a residual interpreter.

A useful heuristic to obtain an implementation function e is to add an extra argument to the semantics m such that this argument is available at compile-time. Shifting work from run-time to compile-time is essential to generate realistic code.

Diacritical Convention When making proofs about two semantic definitions we are frequently required to consider and to compare pairs of values, one from each definition, that are both called by the same name. The diacritical convention as proposed by Stoy [28] is a convenient and systematic way of distinguishing such values. All names from the one definition are given *acute* accents (´) while the names belonging to the other get *grave* accents (`). By convention acute accents are used for decorating the more 'abstract' semantics, while concrete, more to the 'ground' semantics get grave accents.

4 A simple imperative language $\mathcal{W}$

The above theory is put into practice by deriving an implementation of a simple imperative language $\mathcal{W}$. First the syntax and the initial, direct, semantics for $\mathcal{W}$ are introduced. As a first improvement this direct semantics is transformed into a continuation semantics. Next we investigate the efficient compilation of expressions. Arithmetic expressions are translated into conventional three-address code and boolean expressions are implemented using jumping-code [1]. The language $\mathcal{C}$ of section 8 extends $\mathcal{W}$ with simple first order procedures. Here we give a constructive proof that recursive functions can be implemented using a stack and show how tail recursive calls may be eliminated and replaced by jumps. We conclude with a discussion of problems and future work.

4.1 Syntax

The abstract syntax of $\mathcal{W}$ is given by the grammar

$$\begin{array}{rcl}
\mathsf{P} \in \mathsf{Program} & ::= & \textbf{prog}\ \mathsf{Statement} \\
\mathsf{S}, \mathsf{T} \in \mathsf{Statement} & ::= & \textbf{skip} \\
& | & \mathsf{var} := \mathsf{Expression} \\
& | & \mathsf{Statement}\ ;\ \mathsf{Statement} \\
& | & \textbf{if}\ \mathsf{Expression}\ \textbf{then}\ \mathsf{Statement}\ \textbf{else}\ \mathsf{Statement}\ \textbf{fi}
\end{array}$$

The classes of arithmetic and boolean expressions are defined by the single nonterminal Expression.

$$\begin{array}{rcl}
\mathsf{A}, \mathsf{B}, \mathsf{E}, \mathsf{F} \in \mathsf{Expression} & ::= & \textbf{var}\ \mathsf{Var} \\
& | & \textbf{num}\ \mathsf{Num} \\
& | & \mathsf{Expression} ② \mathsf{Expression}
\end{array}$$

The set of binary operators ② includes boolean operators Ⓐ such as **and** and **or**, arithmetic operators ⊕ like + and ×, and relational operators ⊜ such as = and ≥.

Loops in the *concrete* syntax of $\mathcal{W}$ are assumed to be returned as recursive syntax trees by the parser, that is we consider **while** B **do** S **od** as an abbreviation for μ(λloop.**if** B **then** S ; loop **else skip fi**). Dealing with loops on the level of syntax instead of the level of semantics makes no formal difference (see Schmidt [27]) but does make our derivations easier. With semantic recursion, calculations are disrupted by a fixed point induction argument every time a least fixed point μ is encountered. With syntactic recursion we have taken care of that once and for all by means of theorem (CataMu).

4.2 Meaning functions

The semantic functions $\mathcal{M}[\![\_]\!] \in \mathsf{Program} \to \mathsf{program}$, $\mathcal{E}[\![\_]\!] \in \mathsf{Expression} \to \mathsf{expr}$ and $\mathcal{S}[\![\_]\!] \in \mathsf{Statement} \to \mathsf{stat}$ are given by the following set of mutually recursive catamorphisms, written in a functional programming style.

$$\begin{array}{rcl}
\mathcal{M}[\![\mathbf{prog}\ \mathsf{P}]\!] & = & \mathsf{program}\ \mathcal{S}[\![\mathsf{P}]\!] \\
\\
\mathcal{S}[\![\mathbf{skip}]\!] & = & \mathsf{skip} \\
\mathcal{S}[\![\mathsf{x} := \mathsf{E}]\!] & = & \mathsf{assign}\ (\mathsf{x}, \mathcal{E}[\![\mathsf{E}]\!]) \\
\mathcal{S}[\![\mathsf{S} \,;\, \mathsf{T}]\!] & = & \mathsf{seq}\ (\mathcal{S}[\![\mathsf{S}]\!], \mathcal{S}[\![\mathsf{T}]\!]) \\
\mathcal{S}[\![\mathbf{if}\ \mathsf{B}\ \mathbf{then}\ \mathsf{S}\ \mathbf{else}\ \mathsf{T}\ \mathbf{fi}]\!] & = & \mathsf{cond}\ (\mathcal{E}[\![\mathsf{B}]\!], \mathcal{S}[\![\mathsf{S}]\!], \mathcal{S}[\![\mathsf{T}]\!]) \\
\\
\mathcal{E}[\![\mathbf{var}\ \mathsf{x}]\!] & = & \mathsf{var}\ \mathsf{x} \\
\mathcal{E}[\![\mathbf{num}\ \mathsf{n}]\!] & = & \mathsf{num}\ \mathsf{n} \\
\mathcal{E}[\![\mathsf{E} \textcircled{2} \mathsf{F}]\!] & = & \mathsf{oper}_{\textcircled{2}}\ (\mathcal{E}[\![\mathsf{E}]\!], \mathcal{E}[\![\mathsf{F}]\!])
\end{array}$$

These equations show that the semantic functions $\mathcal{M}[\![\_]\!]$, $\mathcal{S}[\![\_]\!]$ and $\mathcal{E}[\![\_]\!]$ inductively replace each of the constructors of a given program by a corresponding compile-time semantic operation (from the algebras prog, stat respectively expr). These operations assemble the denotation of a construct from the denotations of its components in terms of operations of the run-time semantic algebra. The essence of writing a compiler is the extraction of a suitable compile-time algebra from a fixed run-time algebra, that is, given a set of run-time operations we must define compile-time operations $\mathsf{program}$, skip, assign, ... in terms of those. The recursive structure of the interpreter is uniquely determined by the recursive structure of the abstract syntax and therefore not of real interest.

4.3 Dynamic Semantics

The initial dynamic semantics will be a standard direct semantics [27], based upon the following semantic actions.

Evaluating an expression should yield a number in Num. Since expressions can contain variables, their values must be provided at run-time by means of an environment mapping variables to values. Hence the denotation of expressions is a function of type $\mathsf{e}, \mathsf{f}, \mathsf{a}, \mathsf{b} \in \mathsf{expr} = \mathsf{env} \to \mathsf{Num}$. Corresponding to the three possible kinds of expressions, there are three run-time actions. Instruction $\mathsf{VAR}\ \mathsf{x}$ fetches the value to which variable x is bound. Action $\mathsf{VAL}\ \nu$ directly returns the value ν. In order to evaluate a binary operator ②, the environment must be distributed to its arguments first.

$$\begin{array}{rcl}
\mathsf{VAR}\,\mathsf{x}\,\eta & = & \eta\ \mathsf{x} \\
\mathsf{VAL}\,\nu\,\eta & = & \nu \\
(\mathsf{e}\ \widehat{\textcircled{2}}\ \mathsf{f})\ \eta & = & \mathsf{e}\,\eta \textcircled{2} \mathsf{f}\,\eta
\end{array}$$

When no confusion may arise we will just write ② instead of $\widehat{②}$.

An *environment* $\eta \in env = var \to Num$ carries the dynamic values of variables appearing in an expression. Updating the environment is strict; an assignment statement $x := E$ updates x with the *value* of E, thus if $\mathcal{E}[\![E]\!] = \bot$ the meaning of the statement $\mathcal{S}[\![x := E]\!]$ should be $\bot$ as well. In the initial environment η_0 no variable has a proper value.

$$\begin{aligned}
\eta_0 &= \bot \\
\eta[x := \bot] &= \bot \\
\eta[x := v]\ y &= v, \quad \text{if } x = y \\
&= \eta\ y, \ \text{if } x \neq y
\end{aligned}$$

Statement and program denotation are environment transformers $s, t \in stat$, $p \in program = env \to env$. Sequential composition ; of statements is strict in its first argument as we want $\bot$; t to be $\bot$ regardless of the value of t. The unit of ; is SKIP. An assignment statement updates the environment with the new r-value of its l-value.

$$\begin{aligned}
(s\ ;\ t)\ \eta &= \text{strict}\ t\ (s\ \eta) \\
\text{SKIP}\ \eta &= \eta \\
x := e\ \eta &= \eta[x := e\ \eta] \\
(\text{if } b \text{ then } s \text{ else } t)\ \eta &= s\ \eta, \text{if } b\ \eta = 1 \\
&= t\ \eta, \text{if } b\ \eta = 0
\end{aligned}$$

where strict $f\ \bot = \bot$ and strict $f\ a = f\ a$ if $a \neq \bot$.

Using the above operations, the compile-time actions can be defined as follows

$$\begin{aligned}
\text{progam}\ s &= s \\
\text{skip} &= \text{SKIP} \\
\text{assign}\ (x, e) &= x := e \\
\text{seq}\ (s, t) &= s; t \\
\text{cond}\ (b, s, t) &= \text{if } b \text{ then } s \text{ else } t \\
\text{var}\ x &= \text{VAR}\ x \\
\text{num}\ n &= \text{VAL}\ n \\
\text{oper}_{②}\ (e, f) &= e\ \widehat{②}\ f
\end{aligned}$$

Substituting the above definitions in the general meaning functions yields a standard direct semantics

$$\begin{aligned}
\mathcal{M}[\![\textbf{prog P}]\!] &= \text{program}\ \mathcal{S}[\![P]\!] \\
&= \mathcal{S}[\![P]\!] \\
\mathcal{S}[\![\textbf{skip}]\!] &= \text{skip} \\
&= \text{SKIP} \\
\mathcal{S}[\![x := E]\!] &= \text{assign}\ (x, \mathcal{E}[\![E]\!]) \\
&= x := \mathcal{E}[\![E]\!] \\
\mathcal{S}[\![S\ ;\ T]\!] &= \text{seq}\ (\mathcal{S}[\![S]\!], \mathcal{S}[\![T]\!]) \\
&= \mathcal{S}[\![S]\!]; \mathcal{S}[\![T]\!] \\
\mathcal{S}[\![\textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ fi}]\!] &= \text{cond}\ (\mathcal{E}[\![B]\!], \mathcal{S}[\![S]\!], \mathcal{S}[\![T]\!]) \\
&= \text{if } \mathcal{E}[\![B]\!] \text{ then } \mathcal{S}[\![S]\!] \text{ else } \mathcal{S}[\![T]\!] \\
\mathcal{E}[\![\textbf{var x}]\!] &= \text{var}\ x
\end{aligned}$$

$$\begin{aligned} &= \mathsf{VAR}\,\mathsf{x} \\ \mathcal{E}[\![\mathbf{num\ n}]\!] &= \mathsf{num}\ \mathsf{n} \\ &= \mathsf{VAL}\,\mathsf{n} \\ \mathcal{E}[\![\mathsf{E} \textcircled{2} \mathsf{F}]\!] &= \mathsf{oper}_{\textcircled{2}}\ (\mathcal{E}[\![\mathsf{E}]\!], \mathcal{E}[\![\mathsf{F}]\!]) \\ &= \mathcal{E}[\![\mathsf{E}]\!] \textcircled{2} \mathcal{E}[\![\mathsf{F}]\!] \end{aligned}$$

From the proposed abbreviation for while-loops, we get that

$$\mathcal{S}[\![\mathbf{while}\ \mathsf{B}\ \mathbf{do}\ \mathsf{S}\ \mathbf{od}]\!] \;=\; \mu(\lambda \mathsf{loop}.\mathsf{if}\ \mathcal{E}[\![\mathsf{B}]\!]\ \mathsf{then}\ \mathcal{S}[\![\mathsf{S}]\!];\mathsf{loop}\ \mathsf{else}\ \mathsf{SKIP})$$

To improve the above semantics we only have to change the set op run-time actions and revise the compile-time actions accordingly. The actual semantic functions $\mathcal{M}[\![\_]\!]$, $\mathcal{E}[\![\_]\!]$ and $\mathcal{S}[\![\_]\!]$ remain unchanged.

The compiler obtained by factoring the above direct semantics would simply be the identity, the signatures of the action algebra and of the abstract syntax are the same.

5 Continuation semantics

Invariably every efficiency improving transformation is aimed at making explicit an otherwise implicit evaluation order by the introduction of additional continuation arguments. When aiming for a conventional machine (almost) all evaluation order must be explicit. Inventing the proper new semantic domain results from careful study of the interpreter at hand and from that the injection of the old into the new domain usually follows from typing considerations alone.

The first continuation that we will introduce makes control-flow in sequencing of statements explicit. Statement denotations $\acute{s} \in \mathsf{st\acute{a}t} = \mathit{env} \to \mathit{env}$ will be replaced by functions that take their successor statements as an extra argument $\grave{s} \in \mathsf{st\grave{a}t} = (\mathit{env} \to \mathit{env}) \to (\mathit{env} \to \mathit{env})$. Type considerations strongly suggest to define $\mathsf{Stat} \in \mathsf{st\acute{a}t} \to \mathsf{st\grave{a}t}$ as

$$\mathsf{Stat}\ \acute{s}\ \acute{t} \;=\; \acute{s}\,;\acute{t} \qquad \text{(Stat)}$$

The left-inverse Stat^{-1} of Stat takes a concrete statement back into an abstract one

$$\grave{s} = \mathsf{Stat}\ \acute{s} \;\Rightarrow\; \acute{s} = \mathsf{Stat}^{-1}\ \grave{s}$$

and is constructed using the following argument:

$$\begin{aligned} & \acute{s} \\ = \quad & \text{aim at folding } \mathsf{Stat} \\ & \acute{s}\,;\mathsf{id} \\ = \quad & \text{fold } \mathsf{Stat} \\ & \mathsf{Stat}\ \acute{s}\ \mathsf{id} \\ = \quad & \text{assume } \grave{s} = \mathsf{Stat}\ \acute{s} \\ & \grave{s}\ \mathsf{id} \\ = \quad & \text{define } \mathsf{Stat}^{-1}\ \grave{s} = \grave{s}\ \mathsf{id} \\ & \mathsf{Stat}^{-1}\ \grave{s} \end{aligned}$$

Using the fact that Stat is injective we can transform our previous direct semantics into an equivalent continuation semantics.

$$
\begin{array}{cl}
 & \grave{\mathcal{M}}[\![\mathbf{prog}\ S]\!] \\
= & \quad \text{meaning of programs must remain the same} \\
 & \acute{\mathcal{M}}[\![\mathbf{prog}\ S]\!] \\
= & \quad \text{unfold} \\
 & \acute{\mathcal{S}}[\![S]\!] \\
= & \quad \mathsf{Stat}^{-1}\ (\mathsf{Stat}\ s) = s \\
 & \mathsf{Stat}^{-1}\ (\mathsf{Stat}\ \acute{\mathcal{S}}[\![S]\!]) \\
= & \quad \text{unfold} \\
 & \mathsf{Stat}\ \acute{\mathcal{S}}[\![S]\!]\ \mathsf{id} \\
= & \quad \text{assume } \mathsf{Stat}\ \acute{\mathcal{S}}[\![S]\!]\ s = \grave{\mathcal{S}}[\![S]\!]\ s \text{ with } s \text{ static } (*) \\
 & \grave{\mathcal{S}}[\![S]\!]\ \mathsf{id} \\
= & \quad \text{extract } \mathsf{pr\grave{o}gram}\ s = s\ \mathsf{id} \\
 & \mathsf{pr\grave{o}gram}\ \grave{\mathcal{S}}[\![S]\!]
\end{array}
$$

In step (*) we have assumed that $\mathsf{Stat}\ \acute{\mathcal{S}}[\![S]\!]\ s = \grave{\mathcal{S}}[\![S]\!]\ s$ holds with s static. Instantiating the fusion law for $\acute{\mathcal{S}}[\![\_]\!]$ with all static arguments gives sufficient conditions for (*) to hold.

$$
\frac{\begin{array}{l}
\bullet\ \mathsf{Stat}\ \text{strict} \\
\bullet\ \mathsf{Stat}\ \mathsf{sk\acute{\imath}p}\ s = \mathsf{sk\grave{\imath}p}\ s \\
\bullet\ \mathsf{Stat}\ (\mathsf{ass\acute{\imath}gn}\ (x,e))\ s = \mathsf{ass\grave{\imath}gn}\ (x,e)\ s \\
\bullet\ \mathsf{Stat}\ (\mathsf{s\acute{e}q}\ (s,t))\ u = \mathsf{s\grave{e}q}\ (\mathsf{Stat}\ s, \mathsf{Stat}\ t)\ u \\
\bullet\ \mathsf{Stat}\ (\mathsf{c\acute{o}nd}\ (b,s,t))\ u = \mathsf{c\grave{o}nd}\ (b, \mathsf{Stat}\ s, \mathsf{Stat}\ t)\ u
\end{array}}{\mathsf{Stat}\ \acute{\mathcal{S}}[\![S]\!]\ s = \grave{\mathcal{S}}[\![S]\!]\ s}
$$

Strictness of Stat is obvious and our next task now is to find concrete versions $\mathsf{sk\grave{\imath}p}$, $\mathsf{ass\grave{\imath}gn}$, ... of the compile-time actions $\mathsf{sk\acute{\imath}p}$, $\mathsf{ass\acute{\imath}gn}$, ... such that the premise of the fusion law becomes true. As a side-effect of this, new run-time operations will be synthesized where necessary and thus (implicitly) a new transformer from the run-time algebra to the compile-time algebra is built.

The calculations that establish the antecedent of the fusion law consist of a *specialization* part (usually written in the left column) and an *evaluation* part (written in the right column). In a specialization we are only allowed to use static values. When compile-time specialization sticks, we add the dynamic arguments (by η-expansion) and continue (run-time) evaluation until specialization (by η-reduction) can be resumed.

The simplest case is $\mathsf{sk\acute{\imath}p}$, where we find that $\mathsf{Stat}\ \mathsf{sk\acute{\imath}p}\ s = \mathsf{sk\grave{\imath}p}\ s$ holds if we define $\mathsf{sk\grave{\imath}p}\ s = s$

$$
\begin{array}{cl@{\qquad\qquad\qquad}cl}
 & \mathsf{Stat}\ \mathsf{sk\acute{\imath}p}\ s & & \\
= & \quad \text{unfold} & & \\
 & \mathrm{SK\acute{I}P}\ ;\ s & & (\mathrm{SK\acute{I}P}\ ;\ s)\ \eta \\
= & \quad \text{abutting evaluation} & = & \quad \text{evaluate} \\
 & s & & s\ \eta \\
= & \quad \text{extract } \mathsf{sk\grave{\imath}p}\ s = s & & \\
 & \mathsf{sk\grave{\imath}p}\ s & &
\end{array}
$$

Because continuation s is static we have successfully compiled away all SKIP instructions. Had s been dynamic the best we can do is synthesizing a new run-time operation SKÌP.

Stat skíp
= abutting evaluation
SKÌP
= extract skìp = SKÌP
skìp

Stat skíp s η
= evaluate
(SKÍP ; s) η
= evaluate
s η
= synthesize SKÌP s η = s η
SKÌP s η

Assignments cannot be evaluated fully static, and a new run-time operation results to deal with the extra continuation.

Stat (assígn (x, e)) s
= unfold
x :≐ e ; s
= abutting evaluation
x :≐ e s
= extract assìgn (x, e) s = x :≐ e s
assìgn (x, e) s

(x :≐ e ; s) η
= evaluate
s η[x := e η]
= synthesize x :≐ e s η = s η[x := e η]
x :≐ e s η

Hence Stat (assígn (x, e)) s equals assìgn (x, e) s if we define the new compile-time operation assìgn (x, e) s = x :≐ e s in terms of the new instruction x :≐ e s η = s η[x := e η].

Sequential composition eliminates sequential composition of statements ;.

Stat (séq (s, t)) u
= unfold
s; t; u
= fold twice
Stat s (Stat t u)
= extract sèq (s, t) u = s (t u)
sèq (Stat s, Stat t) u

For conditionals we use the fact that composition distributes over choice

$$(\text{if b then s else t}); \text{u} \quad = \quad \text{if b then s}; \text{u else t}; \text{u} \qquad \text{(Cond)}$$

to move the continuation into the two branches of the **if**-statement. Property (Cond) is a free theorem for if _ then _ else _.

Stat (cónd (b, s, t)) u
= unfold
(if b then s else t); u
= (Cond)
if b then s; u else t; u
= fold twice
if b then Stat s u else Stat t u
= extract cònd (b, s, t) u = if b then s u else t u

$\grave{\mathsf{cond}}\ (\mathsf{b}, \mathsf{Stat\ s}, \mathsf{Stat\ t})\ \mathsf{u}$

Applying the fusion law and unfolding the compile-time operations skip, assign, ... results in a continuation semantics for $\mathcal{W}$ that is the same as the semantics given by Schmidt [27] for a similar language.

$$
\begin{aligned}
\mathcal{M}[\![\textbf{prog}\ \mathsf{S}]\!] &= \mathcal{S}[\![\mathsf{S}]\!]\ \mathsf{id}\\
\mathcal{S}[\![\textbf{skip}]\!]\ \mathsf{s} &= \mathsf{s}\\
\mathcal{S}[\![\mathsf{x} := \mathsf{E}]\!]\ \mathsf{s} &= \mathsf{x} := \mathcal{E}[\![\mathsf{E}]\!]\ \mathsf{s}\\
\mathcal{S}[\![\mathsf{S}\ ;\ \mathsf{T}]\!]\ \mathsf{u} &= \mathcal{S}[\![\mathsf{S}]\!]\ (\mathcal{S}[\![\mathsf{T}]\!]\ \mathsf{u})\\
\mathcal{S}[\![\textbf{if}\ \mathsf{B}\ \textbf{then}\ \mathsf{S}\ \textbf{else}\ \mathsf{T}\ \textbf{fi}]\!]\ \mathsf{u} &= \mathsf{if}\ \mathcal{E}[\![\mathsf{B}]\!]\ \mathsf{then}\ \mathcal{S}[\![\mathsf{S}]\!]\ \mathsf{u}\ \mathsf{else}\ \mathcal{S}[\![\mathsf{T}]\!]\ \mathsf{u}\\
\mathcal{E}[\![\textbf{var}\ \mathsf{x}]\!] &= \mathsf{VAR\,x}\\
\mathcal{E}[\![\textbf{num}\ \mathsf{n}]\!] &= \mathsf{VAL\,n}\\
\mathcal{E}[\![\mathsf{E} \textcircled{2} \mathsf{F}]\!] &= \mathcal{E}[\![\mathsf{E}]\!] \textcircled{2} \mathcal{E}[\![\mathsf{F}]\!]
\end{aligned}
$$

It is easy to show that the meaning of while-loops is $\mathcal{S}[\![\textbf{while}\ \mathsf{B}\ \textbf{do}\ \mathsf{S}\ \textbf{od}]\!]\ \mathsf{t} = \mu(\lambda \mathsf{loop}.\mathsf{if}\ \mathcal{E}[\![\mathsf{B}]\!]\ \mathsf{then}\ (\mathcal{S}[\![\mathsf{S}]\!]\ \mathsf{loop})\ \mathsf{else}\ \mathsf{t})$.

5.1 Flowcharts

The current semantics can be factored into a compiler that maps programs into flowcharts. A convenient representation of flowcharts can be obtained by breaking a flowchart into basic blocks using GOTO's for loops and conditionals.

$$\mathsf{GOTO\ s}\ \eta \quad = \quad \mathsf{s}\ \eta$$

The meaning of conditional statements (and accordingly loops) is modified as follows to suggest how a derived compiler could generate code for conditionals and loops.

$$
\begin{aligned}
\mathcal{S}[\![\textbf{if}\ \mathsf{B}\ \textbf{then}\ \mathsf{S}\ \textbf{else}\ \mathsf{T}\ \textbf{fi}]\!]\ \mathsf{u} &= \mathsf{if\ not}\ \mathcal{E}[\![\mathsf{B}]\!]\ \mathsf{then\ GOTO\ else\_}\\
&\ \ \mathsf{else}\ \mathcal{S}[\![\mathsf{S}]\!]\ (\mathsf{GOTO\ fi\_})\ \mathsf{where}\\
&\ \ \mathsf{else\_} = \mathcal{S}[\![\mathsf{T}]\!]\ \mathsf{fi\_}\\
&\ \ \mathsf{fi\_} = \mathsf{u}\\
\mathcal{S}[\![\textbf{while}\ \mathsf{B}\ \textbf{do}\ \mathsf{S}\ \textbf{od}]\!]\ \mathsf{u} &= \mathsf{do\_\ where}\\
&\ \ \mathsf{do\_} = \mathsf{if\ not}\ \mathcal{E}[\![\mathsf{B}]\!]\ \mathsf{then\ GOTO\ od\_}\\
&\ \ \mathsf{else}\ \mathcal{S}[\![\mathsf{S}]\!]\ (\mathsf{GOTO\ do\_})\\
&\ \ \mathsf{od\_} = \mathsf{u}
\end{aligned}
$$

6 Expression continuations

The next goal is to make control-flow in the evaluation of expressions explicit. Looking at $\mathcal{E}[\![\mathsf{E} \textcircled{2} \mathsf{F}]\!]\ \eta = \mathcal{E}[\![\mathsf{E}]\!]\ \eta \textcircled{2} \mathcal{E}[\![\mathsf{F}]\!]\ \eta$, we see that the order of evaluating the arguments of $\textcircled{2}$ is not specified. For an actual implementation some order must be chosen, and subsequently intermediate values have to be stored. Often a stack is introduced for this purpose. Explicit naming of intermediate values is not only easier to derive, it also gives better code for modern load-store RISC architectures.

Explicit control-flow can be introduced into the evaluation of expressions by lifting the order of evaluation subexpressions to the statement level.

$$\mathcal{S}[\![\mathsf{x} := \mathsf{E} \textcircled{2} \mathsf{F}]\!] \quad = \quad \mathcal{S}[\![\mathsf{y} := \mathsf{E}\ ;\ \mathsf{z} := \mathsf{F}\ ;\ \mathsf{x} := \mathsf{y} \textcircled{2} \mathsf{z}]\!] \qquad \text{(ExprSimpl)}$$

where y and z are fresh variables.

This suggest that expressions $\acute{expr} = env \to Num$ should be turned into statements $\grave{expr} = (var \times env \to env) \to (env \to env)$ by means of the transformation $Expr \in \acute{expr} \to \grave{expr}$; again the type of $Expr$ leaves little choice but taking

$$Expr\ e\ (x, s) \quad = \quad x := e\ s \qquad \text{(ExprCont)}$$

Although $Expr$ may generate unnecessary assignments such as for $\mathcal{E}\llbracket \mathbf{var}\ x + \mathbf{var}\ y \rrbracket\ (z, s)$, it is often simpler to eliminate these in subsequent optimization phases than to complicate the transformation to deal with such special cases. It is even doubtful whether we can deal with such 'non-homomorphic' optimisations at all.

$$\begin{array}{ll}
 & \acute{\mathcal{M}}\llbracket \mathbf{prog}\ S \rrbracket \\
= & \quad \text{meaning of programs must remain the same} \\
 & \grave{\mathcal{M}}\llbracket \mathbf{prog}\ S \rrbracket \\
= & \quad \text{unfold} \\
 & \acute{\mathcal{S}}\llbracket S \rrbracket\ id \\
= & \quad \text{assume } \acute{\mathcal{S}}\llbracket S \rrbracket\ s = \grave{\mathcal{S}}\llbracket S \rrbracket\ s\ \wedge\ Expr\ \acute{\mathcal{E}}\llbracket E \rrbracket\ (x, s) = \grave{\mathcal{E}}\llbracket E \rrbracket\ (x, s) \\
 & \grave{\mathcal{S}}\llbracket S \rrbracket\ id \\
= & \quad \text{extract } pr\grave{o}gram\ s = s\ id \\
 & pr\grave{o}gram\ \grave{\mathcal{S}}\llbracket S \rrbracket
\end{array}$$

The fusion law gives sufficient conditions to make the assumption hold.

$$\begin{array}{l}
\bullet\ Expr \text{ strict} \\
\bullet\ c\acute{o}nd\ (b, s, t)\ u = c\grave{o}nd\ (Expr\ b, s, t)\ u \\
\bullet\ wh\acute{i}le\ (b, s)\ t = wh\grave{i}le\ (Expr\ b, s)\ t \\
\bullet\ ass\acute{i}gn\ (x, e)\ s = ass\grave{i}gn\ (x, Expr\ e)\ s \\
\bullet\ Expr\ (v\acute{a}r\ y)\ (x, s) = v\grave{a}r\ y\ (x, s) \\
\bullet\ Expr\ (n\acute{u}m\ n)\ (x, s) = n\grave{u}m\ n\ (x, s) \\
\bullet\ Expr\ (o\acute{p}er_{\textcircled{2}}\ (e, f)\ (x, s) = o\grave{p}er_{\textcircled{2}}\ (Expr\ e, Expr\ f)\ (x, s) \\
\hline
\grave{\mathcal{S}}\llbracket S \rrbracket\ s = \acute{\mathcal{S}}\llbracket S \rrbracket\ s\ \wedge\ Expr\ \acute{\mathcal{E}}\llbracket E \rrbracket\ (x, s) = \grave{\mathcal{E}}\llbracket E \rrbracket\ (x, s)
\end{array}$$

Strictness of $Expr$ is vacuous since updating the environment is strict.

The simplest cases in making the premise of the fusion law true are variables and literal values.

$$\begin{array}{ll}
 & Expr\ (v\acute{a}r\ y)\ (x, s) \\
= & \quad \text{unfold} \\
 & x := VAR\ y\ s \\
= & \quad \text{extract } v\grave{a}r\ y\ (x, s) = x := VAR\ y\ s \\
 & v\grave{a}r\ y\ (x, s)
\end{array}
\qquad
\begin{array}{ll}
 & Expr\ (n\acute{u}m\ n)\ (x, s) \\
= & \quad \text{unfold} \\
 & x := VAL\ n\ s \\
= & \quad \text{extract } n\grave{u}m\ n\ (x, s) = x := VAL\ n\ s \\
 & n\grave{u}m\ n\ (x, s)
\end{array}$$

Improving the compilation of binary operators is the reason why we do these calculations in the first place. The calculation is driving towards folding $Expr$ on the subexpressions e and f, thereby using observation (ExprSimpl) as a heuristic. Doing so we find that $Expr\ (o\acute{p}er_{\textcircled{2}}\ (e, f))\ (x, s) = o\grave{p}er_{\textcircled{2}}\ (Expr\ e, Expr\ f)\ (x, s)$ when $o\grave{p}er_{\textcircled{2}}\ (e, f)\ (x, s) = e\ (y, f\ (z, x := y \textcircled{2} z\ s))$.

$$\begin{array}{ll}
 & Expr\ (o\acute{p}er_{\textcircled{2}}\ (e, f))\ (x, s) \\
= & \quad \text{unfold} \\
 & x := (e \textcircled{2} f)\ s \\
= & \quad \text{observation (ExprSimpl), } y \text{ and } z \text{ fresh variables}
\end{array}$$

$$
\begin{aligned}
& y := e\ (z := f\ (x := \mathrm{VAR}\ y \textcircled{2} \mathrm{VAR}\ z\ s)) \\
= & \quad \text{fold twice} \\
& \mathrm{Expr}\ e\ (y, \mathrm{Expr}\ f\ (z, x := \mathrm{VAR}\ y \textcircled{2} \mathrm{VAR}\ z\ s)) \\
= & \quad \text{synthesize } x := y \textcircled{2} z\ s\ \eta = s\ \eta[x := \eta\ y \textcircled{2} \eta\ z] \\
& \mathrm{Expr}\ e\ (y, \mathrm{Expr}\ f\ (z, x := y \textcircled{2} z\ s)) \\
= & \quad \text{extract } o\grave{p}er_{\textcircled{2}}\ (e, f)\ (x, s) = e\ (y, f\ (z, x := y \textcircled{2} z\ s)) \\
& o\grave{p}er_{\textcircled{2}}\ (\mathrm{Expr}\ e, \mathrm{Expr}\ f)\ (x, s)
\end{aligned}
$$

The exclamation "y and z fresh variables" is of course non-functional. Introducing an explicit supply of fresh variables can be put on top of the interpreter in an orthogonal way using for example monads [32]. The following fragment is copied from an actual Gofer implementation of the current interpreter.

```
oper op e f (x,s)
=
   fresh_variable
   'bind' \y -> fresh_variable
   'bind' \z -> f (z, assignOP op x y z s)
   'bind' \f'-> e (y,f')
   'bind' \e'-> result e'
```

In our experience using monads in the actual derivation is not a good idea as it detracts attention from the core of the matter. When writing an actual interpreter however they are indispensable to provide a source of fresh variable names.

The key observation that allows introducing Expr in the interpretation of statements is that we can assign the value of the expression of a selection statement to a fresh variable before testing the value of that variable

$$\mathcal{S}[\![\textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ fi}]\!] \quad = \quad \mathcal{S}[\![x := B \,;\, \textbf{if } (\textbf{var } x) \textbf{ then } S \textbf{ else } T \textbf{ fi}]\!]$$

With this insight the derivation of the new compilation scheme for conditionals is simple enough.

$$
\begin{aligned}
& c\acute{o}nd\ (b, s, t)\ u \\
= & \quad \text{unfold} \\
& \mathrm{if}\ b\ \mathrm{then}\ s\ u\ \mathrm{else}\ t\ u \\
= & \quad \text{eureka, } x \text{ a fresh variable} \\
& x := b\ (\mathrm{if}\ \mathrm{VAR}\ x\ \mathrm{then}\ s\ u\ \mathrm{else}\ t\ u) \\
= & \quad \text{fold} \\
& \mathrm{Expr}\ b\ (x, \mathrm{if}\ \mathrm{VAR}\ x\ \mathrm{then}\ s\ u\ \mathrm{else}\ t\ u) \\
= & \quad \text{extract } c\grave{o}nd\ (b, s, t)\ u = b\ (x, \mathrm{if}\ \mathrm{VAR}\ x\ \mathrm{then}\ s\ u\ \mathrm{else}\ t\ u) \\
& c\grave{o}nd\ (\mathrm{Expr}\ b, s, t)\ u
\end{aligned}
$$

Similarly we find $ass\grave{i}gn\ (x, e)\ s = e\ (x, s)$. Referring to the fusion law we get the new semantics:

$$
\begin{aligned}
\mathcal{M}[\![\textbf{prog } S]\!] &= \mathcal{S}[\![S]\!]\ \mathrm{id} \\
\mathcal{S}[\![\textbf{skip}]\!]\ s &= s \\
\mathcal{S}[\![x := E]\!]\ s &= \mathcal{E}[\![E]\!]\ (x, s) \\
\mathcal{S}[\![\textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ fi}]\!]\ u &= \mathcal{E}[\![B]\!]\ (x, \mathrm{if}\ \mathrm{VAR}\ x\ \mathrm{then}\ \mathcal{S}[\![S]\!]\ u\ \mathrm{else}\ \mathcal{S}[\![S]\!]\ u) \\
\mathcal{E}[\![\textbf{var } y]\!]\ (x, s) &= x := \mathrm{VAR}\ y\ s \\
\mathcal{E}[\![\textbf{num } n]\!]\ (x, s) &= x := \mathrm{VAL}\ n\ s \\
\mathcal{E}[\![E \textcircled{2} F]\!]\ (x, s) &= \mathcal{E}[\![E]\!]\ (y, \mathcal{E}[\![F]\!]\ (z, x := y \textcircled{2} z\ s))
\end{aligned}
$$

The new set of run-time operations is:

$$
\begin{array}{rcl}
x := \mathsf{VAR}\; y\; s\; \eta & = & s\; \eta[x := \eta\; y] \\
x := \mathsf{VAL}\; n\; s\; \eta & = & s\; \eta[x := n] \\
x := y \textcircled{2} z\; s\; \eta & = & s\; \eta[x := \eta\; y \textcircled{2} \eta\; z]
\end{array}
$$

7 Short Circuit Evaluation

Most programming languages, with Algol68 and Pascal being exceptions to the rule, specify *short circuit* evaluation of (certain) boolean expressions. The C manual [15] for example says:

> "Unlike **&**, **&&** guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to 0, the value of the expression is 0. Otherwise the right operand is evaluated, and if it is equal to 0, the expression's value is 0, otherwise 1."

Since our interpreter inherits boolean operators from its meta-language we must make sure that they behave as required. We can kill two birds with one stone by using the following program equivalences to eliminate boolean expressions altogether

$$
\begin{array}{rcl}
\mathcal{S}[\![x := \mathsf{B}]\!] & = & \mathcal{S}[\![\textbf{if } \mathsf{B} \textbf{ then } x := \textbf{true} \textbf{ else } x := \textbf{false} \textbf{ fi}]\!] \\
\mathcal{S}[\![\textbf{if } (\mathsf{A} \textbf{ or } \mathsf{B}) \textbf{ then } \mathsf{S} \textbf{ else } \mathsf{T} \textbf{ fi}]\!] & = & \mathcal{S}[\![\textbf{if } \mathsf{A} \textbf{ then } \mathsf{S} \textbf{ else } (\textbf{if } \mathsf{B} \textbf{ then } \mathsf{S} \textbf{ else } \mathsf{T} \textbf{ fi}) \textbf{ fi}]\!] \\
\mathcal{S}[\![\textbf{if } (\mathsf{A} \textbf{ and } \mathsf{B}) \textbf{ then } \mathsf{S} \textbf{ else } \mathsf{T} \textbf{ fi}]\!] & = & \mathcal{S}[\![\textbf{if } \mathsf{A} \textbf{ then } (\textbf{if } \mathsf{B} \textbf{ then } \mathsf{S} \textbf{ else } \mathsf{T} \textbf{ fi}) \textbf{ else } \mathsf{T} \textbf{ fi}]\!] \\
\mathcal{S}[\![\textbf{if } (\textbf{not } \mathsf{A}) \textbf{ then } \mathsf{S} \textbf{ else } \mathsf{T} \textbf{ fi}]\!] & = & \mathcal{S}[\![\textbf{if } \mathsf{A} \textbf{ then } \mathsf{T} \textbf{ else } \mathsf{S} \textbf{ fi}]\!] \\
\mathcal{S}[\![\textbf{if true then } \mathsf{S} \textbf{ else } \mathsf{T} \textbf{ fi}]\!] & = & \mathsf{S} \\
\mathcal{S}[\![\textbf{if false then } \mathsf{S} \textbf{ else } \mathsf{T} \textbf{ fi}]\!] & = & \mathsf{T}
\end{array}
$$

The duplication of code in the transformations for **and** and **or** can easily be avoided by adding an extra **goto** or so. In practical programs expressions tend not to be very deeply nested however and then the cure might be worse than the disease due to broken pipelines caused by these additional jumps.

If short circuit code is required, boolean connectives have to be translated differently from arithmetical operators. Therefore the syntax is changed to distinguish between arithmetic and boolean expressions.

$$
\begin{array}{rcl}
\mathsf{P} \in \mathit{program} & ::= & \textbf{prog}\ \mathit{statement} \\
\mathsf{S},\mathsf{T} \in \mathit{statement} & ::= & \textbf{skip} \\
 & | & \mathit{var} := \mathit{expression} \\
 & | & \mathit{statement}\ ;\ \mathit{statement} \\
 & | & \textbf{if}\ \mathit{boolean}\ \textbf{then}\ \mathit{statement}\ \textbf{else}\ \mathit{statement}\ \textbf{fi} \\
\mathsf{E},\mathsf{F} \in \mathit{expression} & ::= & \textbf{var}\ \mathit{var} \\
 & | & \textbf{num}\ \mathit{num} \\
 & | & \mathit{expression} \oplus \mathit{expression} \\
 & | & \textbf{bool}\ \mathit{boolean} \\
\mathsf{A},\mathsf{B} \in \mathit{boolean} & ::= & \textbf{var}\ \mathit{var} \\
 & | & \textbf{true} \mid \textbf{false} \\
 & | & \mathit{expression} \textcircled{=} \mathit{expression} \\
 & | & \mathit{boolean} \textcircled{\wedge} \mathit{boolean}
\end{array}
$$

Short circuit evaluation of boolean expressions is introduced by simultaneous application of the transformations:

$$\begin{aligned} \mathrm{Bool}\ b\ (s,t) &= \mathrm{if}\ b\ \mathrm{then}\ s\ \mathrm{else}\ t \\ \mathrm{Expr}\ e\ (x,s) &= (x := e)\ s \end{aligned}$$

to the continuation semantics of §6 (adopted in the obvious way to deal with boolean expressions.)

First we will derive the new semantic operations for boolean expressions. The generated code for a variable dynamically chooses a branch to continue because the value of variable x is dynamic.

$$\begin{aligned} & \mathrm{Bool}\ (\mathrm{v\acute{a}r}\ x)\ (s,t) \\ = \ & \mathrm{if}\ \mathrm{VAR}\, x\ \mathrm{then}\ s\ \mathrm{else}\ t \\ = \ & \mathrm{v\grave{a}r}\ x\ (s,t) \end{aligned}$$

Atomic boolean expressions are eliminated at compile-time, thanks to the fact that the pair of continuations (s,t) is static. The calculation for $\mathrm{f\acute{a}lse}$ is very similar and not shown.

$$\begin{aligned} & \mathrm{Bool}\ \mathrm{tr\acute{u}e}\ (s,t) \\ = \ & \mathrm{if}\ \mathrm{VAL}\ 1\ \mathrm{then}\ s\ \mathrm{else}\ t \\ = \ & s \\ = \ & \mathrm{tr\grave{u}e}\ (s,t) \end{aligned}$$

Complex boolean expressions are reduced to a rat's nest of conditionals as suggested by the above observations. Again, the calculation for $(\mathrm{\acute{o}r}\ (a,b))$ is similar and therefor omitted.

$$\begin{aligned} & \mathrm{Bool}\ (\mathrm{\acute{a}nd}\ (a,b))\ (s,t) \\ = \ & \mathrm{if}\ (a\ \mathrm{AND}\ b)\ \mathrm{then}\ s\ \mathrm{else}\ t \\ = \ & \mathrm{if}\ a\ \mathrm{then}\ (\mathrm{if}\ b\ \mathrm{then}\ s\ \mathrm{else}\ t)\ \mathrm{else}\ t \\ = \ & \mathrm{Bool}\ a\ (\mathrm{Bool}\ b\ (s,t), t) \\ = \ & \mathrm{\grave{a}nd}\ (\mathrm{Bool}\ a, \mathrm{Bool}\ b)\ (s,t) \end{aligned}$$

Relational expressions cannot be encoded by means of control-flow, and thus require somewhat more work.

$$\begin{aligned} & \mathrm{Bool}\ (\mathrm{e\acute{q}ual}\ (e,f))\ (s,t) \\ = \ & \mathrm{if}\ e \ominus f\ \mathrm{then}\ s\ \mathrm{else}\ t \\ = \ & x := (e \ominus f)\ (\mathrm{if}\ \mathrm{VAR}\, x\ \mathrm{then}\ s\ \mathrm{else}\ t) \\ = \ & y := e\ (z := f\ (x := y \ominus z\ (\mathrm{if}\ \mathrm{VAR}\, x\ \mathrm{then}\ s\ \mathrm{else}\ t))) \\ = \ & \mathrm{Expr}\ e\ (y, \mathrm{Expr}\ f\ (z, x := y \ominus z\ (\mathrm{if}\ \mathrm{VAR}\, x\ \mathrm{then}\ s\ \mathrm{else}\ t))) \\ = \ & \mathrm{e\grave{q}ual}\ (\mathrm{Expr}\ e, \mathrm{Expr}\ f)\ (s,t) \end{aligned}$$

Boolean expressions form the interface between Expr and Bool.

$$\begin{aligned} & \mathrm{Expr}\ (\mathrm{bo\acute{o}l}\ b)\ (x,s) \\ = \ & x := b\ s \\ = \ & \mathrm{if}\ b\ \mathrm{then}\ x := \mathrm{VAL}\ 1\ s\ \mathrm{else}\ x := \mathrm{VAL}\ 0\ s \\ = \ & \mathrm{Bool}\ b\ (x := \mathrm{TRUE}\ s, x := \mathrm{FALSE}\ s) \\ = \ & \mathrm{bo\grave{o}l}\ (\mathrm{Bool}\ b)\ (x,s) \end{aligned}$$

The remaining cases for Expr remain unchanged with respect to the previous calculation of expression continuations, but conditionals now contain boolean instead of integer expressions.

$$
\begin{array}{ll}
 & \acute{\mathsf{cond}}\ (b, s, t)\ u \\
= & \mathsf{if}\ b\ \mathsf{then}\ s\ u\ \mathsf{else}\ t\ u \\
= & \mathsf{Bool}\ b\ (s\ u, t\ u) \\
= & \grave{\mathsf{cond}}\ (\mathsf{Bool}\ b, s, t)\ u
\end{array}
$$

The fusion law yields the new semantics.

$$
\begin{array}{rcl}
\mathcal{M}[\![\mathbf{prog}\ S]\!] & = & \mathcal{S}[\![S]\!]\ id \\
\\
\mathcal{S}[\![\mathbf{skip}]\!]\ s & = & s \\
\mathcal{S}[\![x := E]\!]\ S & = & \mathcal{E}[\![E]\!]\ (x, s) \\
\mathcal{S}[\![S\ ;\ T]\!]\ u & = & \mathcal{S}[\![S]\!]\ (\mathcal{S}[\![T]\!]\ u) \\
\mathcal{S}[\![\mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{fi}]\!]\ u & = & \mathcal{B}[\![B]\!]\ (\mathcal{S}[\![S]\!]\ u, \mathcal{S}[\![T]\!]\ u) \\
\mathcal{S}[\![\mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}]\!]\ t & = & \mu(\lambda loop.\mathcal{B}[\![B]\!]\ (\mathcal{S}[\![S]\!]\ loop, t)) \\
\\
\mathcal{E}[\![\mathbf{var}\ y]\!]\ (x, s) & = & y := \mathsf{VAR}\ x\ s \\
\mathcal{E}[\![\mathbf{num}\ n]\!]\ (x, s) & = & x := \mathsf{VAL}\ n\ s \\
\mathcal{E}[\![E \oplus F]\!]\ (z, s) & = & \mathcal{E}[\![E]\!]\ (x, \mathcal{E}[\![F]\!]\ (y, z := x \oplus y\ s)) \\
\mathcal{E}[\![\mathbf{bool}\ B]\!]\ (z, s) & = & \mathcal{B}[\![B]\!]\ (z := \mathsf{VAL}\ 1\ s, z := \mathsf{VAL}\ 0\ s) \\
\\
\mathcal{B}[\![\mathbf{var}\ x]\!]\ (s, t) & = & \mathsf{if}\ \mathsf{VAR}\ x\ \mathsf{then}\ s\ \mathsf{else}\ t \\
\mathcal{B}[\![\mathbf{true}]\!]\ (s, t) & = & s \\
\mathcal{B}[\![\mathbf{false}]\!]\ (s, t) & = & t \\
\mathcal{B}[\![E \ominus F]\!]\ (s, t) & = & \mathcal{E}[\![E]\!]\ (y, \mathcal{E}[\![F]\!]\ (z, x := y \ominus z\ (\mathsf{if}\ \mathsf{VAR}\ x\ \mathsf{then}\ s\ \mathsf{else}\ t))) \\
\mathcal{B}[\![A\ \mathbf{and}\ B]\!]\ (s, t) & = & \mathcal{B}[\![A]\!]\ (\mathcal{B}[\![B]\!]\ (s, t), t) \\
\mathcal{B}[\![A\ \mathbf{or}\ B]\!]\ (s, t) & = & \mathcal{B}[\![A]\!]\ (s, \mathcal{B}[\![B]\!]\ (s, t))
\end{array}
$$

It is hard to imagine that anyone can write this interpreter from scratch. A Gofer implementation of the compiler generated from the above semantics using theorem (Factor) compiles a program for computing factorials

```
n := num ... ;
fac := num 1 ;
while (var n > num 0)
do
  fac := var n × var fac ;
  n := var n − num 1
od ;
...
```

into the following fragment of pidgin C (after some trivial reformatting)

```
    n = ...;
    fac = 1;
do:
    x6 = n;
    x7 = 0;
    x8 = x6 > x7;
    if(!x8){goto od;};
    x5 = n;
    x4 = fac;
    fac = x5 * x4;
    x3 = n;
    x2 = 1;
    n = x3 - x2;
    goto do;
od:
    ...
    exit (0);
```

8 Adding Procedures: from $\mathcal{W}$ to $\mathcal{C}$

Now that we have seen how a realistic compiler for $\mathcal{W}$ can be derived in a few simple steps, we extend the language with first order procedures. Function procedures are added to $\mathcal{W}$ by extending the syntax with clauses for returning a value, for calling a procedure and for procedures bodies.

$$\begin{aligned} \mathit{statement} &::= \ldots \mid \textbf{return}\ \mathit{expression} \\ \mathit{expression} &::= \ldots \mid \textbf{call}\ \mathit{procedure}\ (\mathit{expression}) \\ \mathsf{P} \in \mathit{procedure} &::= \textbf{proc}\ \mathit{var}\ \textbf{begin}\ \mathit{statement}\ \textbf{end} \end{aligned}$$

Recursive procedures will be represented by cyclic programs. An example recursive procedure is fac

```
μ(λfac  .  proc n
           begin
             if n = 0 then return 1
             else return n * call fac (n − 1)
             fi
           end)
```

Usually recursive procedures are assumed to be finite and recursion is solved semantically by means of a symbol table mapping procedure names into their denotations. Using cyclic programs it is impossible to describe dynamic binding. Syntactic recursion is static binding at its extreme.

8.1 Continuation Semantics

Statement continuations will get type $\mathit{env} \to \mathsf{Num}$ instead of $\mathit{env} \to \mathit{env}$ because procedures return a value. The CALL-instruction $x := \mathsf{CALL}\ ((p, y), z)$ calls procedure p with an initial environment in which the formal argument y is bound to the value of the actual argument z. The value computed by p $(y := z)$ is assigned to x.

$$\begin{aligned} \mathsf{RETURN}\ x\ s\ \eta &= \eta\ x \\ x := \mathsf{CALL}\ ((p, y), z)\ s\ \eta &= s\ \eta[x := v]\ \mathit{where}\ v = p\ \eta_0[y := \eta\ z] \\ \mathsf{EXIT}\ s\ \eta &= \bot \end{aligned}$$

The extra valuation functions for the new syntactic elements are defined using the new semantic operations. Besides $\mathcal{M}[\![\_]\!]$ all other semantic functions remain unchanged.

$$\mathcal{M}[\![\textbf{prog}\ \mathsf{S}]\!] = \mathcal{S}[\![\mathsf{S}]\!]\ (\mathsf{EXIT}\ \bot)$$

$$\begin{aligned}
\mathcal{S}[\![\textbf{return } \mathsf{E}]\!]\ \mathsf{s} &= \mathcal{E}[\![\mathsf{E}]\!]\ (\mathsf{x}, \mathsf{RETURN}\ \mathsf{x}\ \mathsf{s}) \\
\mathcal{E}[\![\textbf{call } \mathsf{P}(\mathsf{E})]\!]\ (\mathsf{x}, \mathsf{s}) &= \mathcal{E}[\![\mathsf{E}]\!]\ (\mathsf{y}, \mathsf{x} := \mathsf{CALL}\ (\mathcal{P}[\![\mathsf{P}]\!], \mathsf{y})\ \mathsf{s}) \\
\mathcal{P}[\![\_]\!] &\in \mathit{procedure} \to \mathit{stat} \times \mathit{var} \\
\mathcal{P}[\![\textbf{proc } \mathsf{x} \textbf{ begin } \mathsf{S} \textbf{ end}]\!] &= (\mathcal{S}[\![\mathsf{S}]\!]\ (\mathsf{EXIT}\ \bot), \mathsf{x})
\end{aligned}$$

A procedure or program that does not RETURN explicitly, implicitly returns $\bot$.

8.2 Introducing the dump

The semantics of a procedure call

$$\mathsf{x} := \mathsf{CALL}\ ((\mathsf{p}, \mathsf{y}), \mathsf{z})\ \mathsf{s}\ \eta \quad = \quad \mathsf{s}\ \eta[\mathsf{x} := \nu]\ \mathit{where}\ \nu = \mathsf{p}\ \eta_0[\mathsf{y} := \eta\ \mathsf{z}]$$

does not reflect the standard subroutine call, where evaluation on the caller's side is temporarily suspended and control is transferred from caller to callee which eventually returns its result to back to the caller. This implicit evaluation order will be explicated by introducing yet another continuation, the *dump*. The dump continuation represents the suspended computation of the caller of the currently executing procedure, it has type $\mathit{dump} = \mathsf{Num} \to \mathsf{Num}$, and should be strict. Given the result of the callee, the caller may resume computing its result, but if the callee evaluates to $\bot$ the whole computation has to fail.

The injective function $\mathsf{Dump} \in (\mathit{env} \to \mathsf{Num}) \to (\mathit{env} \to \mathit{dump} \to \mathsf{Num})$ maps an abstract continuation $\acute{s}$ that does not expect a dump, into a concrete one $\grave{s} = \mathsf{Dump}\ \acute{s}$ that does expect a dump. It sounds like a cliche, but again the type of Dump leads us to the definition

$$\mathsf{Dump}\ \acute{s}\ \eta\ \delta \quad = \quad \delta\ (\acute{s}\ \eta)$$

The left-inverse $\mathsf{Dump}^{-1} \in (\mathit{env} \to \mathit{dump} \to \mathsf{Num}) \to (\mathit{env} \to \mathsf{Num})$ maps a concrete continuation $\grave{s} = \mathsf{Dump}\ \acute{s}$ back into an abstract one $\acute{s} = \mathsf{Dump}^{-1}\ \grave{s}$.

$$\begin{aligned}
& \mathsf{Dump}^{-1}\ \grave{s}\ \eta \\
= \quad & \qquad \grave{s} = \mathsf{Dump}\ \acute{s} \\
& \mathsf{Dump}^{-1}\ (\mathsf{Dump}\ \acute{s})\ \eta \\
= \quad & \qquad \text{meaning of programs must remain unchanged} \\
& \acute{s}\ \eta \\
= \quad & \qquad \text{aiming at folding Dump} \\
& \mathsf{id}\ (\acute{s}\ \eta) \\
= \quad & \qquad \text{fold} \\
& \mathsf{Dump}\ \acute{s}\ \eta\ \mathsf{id} \\
= \quad & \qquad \mathsf{Dump}\ \acute{s} = \grave{s} \\
& \grave{s}\ \eta\ \mathsf{id}
\end{aligned}$$

Thus $\mathsf{Dump}^{-1}\ \grave{s}\ \eta = \grave{s}\ \eta\ \mathsf{id}$ is a left-inverse of Dump.

The new semantics is calculated using the fact that $\mathsf{Dump}^{-1}\ (\mathsf{Dump}\ \mathsf{s}) = \mathsf{s}$ for $\mathsf{s} \in \mathit{env} \to \mathsf{Num}$.

$$\begin{aligned}
& \grave{\mathcal{M}}[\![\textbf{prog } \mathsf{S}]\!] \\
= \quad & \qquad \text{wish} \\
& \acute{\mathcal{M}}[\![\textbf{prog } \mathsf{S}]\!] \\
= \quad & \qquad \text{unfold}
\end{aligned}$$

$\acute{\mathcal{S}}[\![\text{S}]\!]$ (EXIT ⊥)
= $\quad$ $\acute{\mathcal{S}}[\![\text{S}]\!]$ s = Dump^{-1} (Dump $\acute{\mathcal{S}}[\![\text{S}]\!]$ s)
Dump^{-1} (Dump ($\acute{\mathcal{S}}[\![\text{S}]\!]$ (EXIT ⊥)))
= $\quad$ assume Dump ($\acute{\mathcal{S}}[\![\text{S}]\!]$ s) = $\grave{\mathcal{S}}[\![\text{S}]\!]$ (Dump s)
Dump^{-1} ($\grave{\mathcal{S}}[\![\text{S}]\!]$ (Dump (EXIT ⊥)))
= $\quad$ extract prògram s = Dump^{-1} (s (EXIT ⊥))
prògram $\grave{\mathcal{S}}[\![\text{S}]\!]$

The assumption follows from the instance of higher-order fusion given below where Proc (s, x) = (Dump s, x)

- Dump strict
- Dump (skíp s) = skìp (Dump s)
- Dump (séq (ś, t́) u) = sèq (s̀, t̀) (Dump u)
  ⇐ Dump (ś u) = s̀ (Dump u) ∧ Dump (t́ u) = t̀ (Dump u)
- Dump (assígn (x, é) s) = assìgn (x, è) (Dump s)
  ⇐ Dump (é (x, s)) = è (x, Dump s)
- Dump (cónd (b́, ś, t́) u) = cònd (b̀, s̀, t̀) (Dump u)
  ⇐ Dump (b́ (x, s)) = b̀ (x, Dump s) ∧
  Dump (ś u) = s̀ (Dump u) ∧ Dump (t́ u) = t̀ (Dump u)
- Dump (oṕer$_{②}$ (é, f́) (x, s)) = op̀er$_{②}$ (è, f̀) (x, Dump s)
  ⇐ Dump (é (x, s)) = è (x, Dump s) ∧ Dump (f́ (x, s)) = f̀ (x, Dump s)
- Dump (cáll (ṕ, é) (x, u)) = càll (p̀, è) (x, Dump u)
  ⇐ Dump (é (x, u)) = è (x, Dump u) ∧ Proc ṕ = p̀
- Proc (próc (ś, x)) = pròc (s̀, x)
  ⇐ Dump (ś u) = s̀ (Dump u)

Dump ($\acute{\mathcal{S}}[\![\text{S}]\!]$ s) = $\grave{\mathcal{S}}[\![\text{S}]\!]$ (Dump s)∧
Dump ($\acute{\mathcal{E}}[\![\text{E}]\!]$ (s, x)) = $\grave{\mathcal{E}}[\![\text{E}]\!]$ (x, Dump s)∧
Proc $\acute{\mathcal{P}}[\![\text{P}]\!]$ = $\grave{\mathcal{P}}[\![\text{P}]\!]$

Now we can try to find concrete compile-time operations that make the fusion law hold. The nested premisses will be referred to as (IH) (induction hypothesis). The meaning of **skip** statements and sequencing remains unchanged.

Dump (skíp s)
= $\quad$ unfold
Dump s
= $\quad$ extract skìp s = s
skìp (Dump s)

Dump ((séq (ś, t́) u)
= $\quad$ unfold
Dump (ś (t́ u))
= $\quad$ IH twice
s̀ (t̀ (Dump u))
= $\quad$ extract sèq (s, t) u = s (t u)
sèq (s̀, t̀) (Dump u)

If we had used normal fusion Dump (skíp s) = skìp s instead, the result is a correct but operationally unsatisfying semantics.

Dump (skíp s) η δ
= $\quad$ unfold
δ (s η)
= $\quad$ synthesize SKIP s η δ = δ (s η)

SKIP s η δ
= extract
skìp s η δ

It forces the introduction of a weird run-time instruction SKIP that returns immediately. It took the author a long time to realize that higher order fusion was the way to go.

We continue our derivation with the assignment statement, where we find that Dump (assígn (x, é) s) = assìgn (x, è) (Dump s) if assìgn (x, e) s = e (x, s).

Dump (assígn (x, é) s)
= unfold
Dump (é (x, s))
= IH
è (x, Dump s)
= extract
assìgn (x, è) (Dump s)

The side effect of applying Dump to RETURN-statements is a new instruction RETÙRN x s η δ = δ (η x).

Dump (retúrn é s)
= unfold
Dump (é (x, RETÚRN x s))
= IH
è (x, Dump (RETÚRN x s))
= abutting evaluation
è (x, RETÙRN x (Dump s))
= extract retùrn
retùrn è (Dump s)

Dump (RETÚRN x s) η δ
= unfold
δ (RETÚRN x s η)
= evaluate
δ (η x)
= synthesize RETÙRN
RETÙRN x (Dump s) η δ

The instruction RETÙRN x s η δ = δ (η x) captures the intuition of the statement **return** E, namely return the value of E to the caller of the caller.

Conditionals pose no particular problems and hence they are omitted.

Having shown that Dump promotes over all operations of $\mathcal{S}[\![\_]\!]$, we now must show that Dump also promotes over the operations of $\mathcal{E}[\![\_]\!]$.

$$\begin{aligned}
& \mathrm{Dump}\ (\mathrm{o\acute{p}er}_{\textcircled{2}}\ (\acute{a},\acute{b})\ (x,s)) \\
= \quad & \text{unfold} \\
& \mathrm{Dump}\ (\acute{a}\ (y,\acute{b}\ (z,x \mathrel{:\acute{=}} y \textcircled{2} z\ s))) \\
= \quad & \text{IH twice} \\
& \grave{a}\ (y,\grave{b}\ (z,\mathrm{Dump}\ (x \mathrel{:\acute{=}} y \textcircled{2} z\ s))) \\
= \quad & \text{abutting evaluation} \\
& \grave{a}\ (y,\grave{b}\ (z,x \mathrel{:\grave{=}} y \textcircled{2} z\ (\mathrm{Dump}\ s))) \\
= \quad & \text{extract} \\
& \mathrm{o\grave{p}er}_{\textcircled{2}}\ (\grave{a},\grave{b})\ (x,\mathrm{Dump}\ s)
\end{aligned}$$

$$\begin{aligned}
& \mathrm{Dump}\ (x \mathrel{:\acute{=}} y \textcircled{2} z\ s)\ \eta\ \delta \\
= \quad & \text{unfold} \\
& \delta\ (x \mathrel{:\acute{=}} y \textcircled{2} z\ s\ \eta) \\
= \quad & \text{evaluate} \\
& \delta\ (s\ \eta[x := \eta\ y \textcircled{2} \eta\ z]) \\
= \quad & \text{fold} \\
& \mathrm{Dump}\ s\ \eta[x := \eta\ y \textcircled{2} \eta\ z]\ \delta \\
= \quad & \text{synthesize} \\
& x \mathrel{:\grave{=}} y \textcircled{2} z\ (\mathrm{Dump}\ s)\ \eta\ \delta
\end{aligned}$$

For expressions compiled into VAL and VAR we find new instructions in a similar fashion.

$$\begin{aligned}
x := \mathrm{VAR}\, y\ s\ \eta\ \delta &= s\ \eta[x := \eta\ y]\ \delta \\
x := \mathrm{VAL}\, n\ s\ \eta\ \delta &= s\ \eta[x := n]\ \delta
\end{aligned}$$

The most difficult case is the procedure call.

$$\begin{aligned}
& \mathrm{Dump}\ (\mathrm{c\acute{a}ll}\ ((\acute{p},y),\acute{e})\ (x,s)) \\
= \quad & \text{unfold} \\
& \mathrm{Dump}\ (\acute{e}\ (z,x \mathrel{:\acute{=}} \mathrm{CALL}\ ((\acute{p},y),z)\ s)) \\
= \quad & \text{IH} \\
& \grave{e}\ (z,\mathrm{Dump}\ (x \mathrel{:\acute{=}} \mathrm{CALL}\ ((\acute{p},y),z)\ s)) \\
= \quad & \text{evaluation, see below} \\
& \grave{e}\ (z,x \mathrel{:\grave{=}} \mathrm{CALL}\ (\mathrm{Proc}\ ((\acute{p},y),z))\ (\mathrm{Dump}\ s)) \\
= \quad & \text{IH} \\
& \grave{e}\ (z,x \mathrel{:\grave{=}} \mathrm{CALL}\ ((\grave{p},y),z)\ (\mathrm{Dump}\ s)) \\
= \quad & \text{extract} \\
& \mathrm{c\grave{a}ll}\ ((\grave{p},y),\grave{e})\ s
\end{aligned}$$

In the fourth step of this calculation the assumption has been made that $\mathrm{Dump}\ (x \mathrel{:\acute{=}} \mathrm{CALL}\ ((\acute{p},y),z))\ s = x \mathrel{:\grave{=}} \mathrm{CALL}\ (\mathrm{Proc}\ ((\acute{p},y),z))\ (\mathrm{Dump}\ s)$, this remains to be shown.

$$\begin{aligned}
& \mathrm{Dump}\ (x \mathrel{:\acute{=}} \mathrm{CALL}\ ((p,y),z))\ s\ \eta\ \delta \\
= \quad & \text{unfold} \\
& \delta\ (x \mathrel{:\acute{=}} \mathrm{CALL}\ ((p,y),z)\ s\ \eta) \\
= \quad & \text{evaluate} \\
& \delta\ (s\ \eta[x := v]\ \text{where}\ v = p\ \eta_0[y := \eta\ z]) \\
= \quad & \text{lambda calculus} \\
& \delta\ (s\ \eta[x := v])\ \text{where}\ v = p\ \eta_0[y := \eta\ z] \\
= \quad & \text{lambda calculus} \\
& (\lambda v.\delta\ (s\ \eta[x := v]))\ (p\ \eta_0[y := \eta\ z]) \\
= \quad & \text{fold} \\
& \mathrm{Dump}\ p\ \eta_0[y := \eta\ z]\ (\lambda v.\mathrm{Dump}\ s\ \eta[x := v]\ \delta) \\
= \quad & \text{synthesize} \\
& x \mathrel{:\grave{=}} \mathrm{CALL}\ (\mathrm{Proc}\ ((p,y),z)\ (\mathrm{Dump}\ s)\ \eta\ \delta
\end{aligned}$$

We are nearly done, the last remaining case is $\mathsf{Proc}\ (\mathsf{pr\acute{o}c}\ (s,x))$.

$$\begin{array}{cl}
 & \mathsf{Proc}\ (\mathsf{pr\acute{o}c}\ (\acute{s},x)) \\
= & \quad \text{unfold} \\
 & \mathsf{Proc}\ (\acute{s}\ (\mathsf{EX\acute{I}T}\ \bot),x) \\
= & \quad \text{IH} \\
 & (\grave{s}\ (\mathsf{Dump}\ (\mathsf{EX\acute{I}T}\ \bot)),x) \\
= & \quad \text{abutting evaluation} \\
 & (\grave{s}\ (\mathsf{EX\grave{I}T}\ \bot),x) \\
= & \quad \text{extract} \\
 & \mathsf{pr\grave{o}c}\ (\grave{s},x)
\end{array}
\qquad\qquad
\begin{array}{cl}
 & \mathsf{Dump}\ (\mathsf{EX\acute{I}T}\ s)\ \eta\ \delta \\
= & \quad \text{unfold} \\
 & \delta\ (\mathsf{EX\grave{I}T}\ s\ \eta) \\
= & \quad \text{evaluate} \\
 & \delta\ \bot \\
= & \quad \text{synthesize} \\
 & \mathsf{EX\grave{I}T}\ s\ \eta\ \delta
\end{array}$$

Applying higher order fusion yields the final semantics for $\mathcal{C}$.

$$\begin{array}{rcl}
\mathcal{M}\llbracket\textbf{prog}\ \mathsf{P}\rrbracket & = & \mathsf{Dump}^{-1}\ (\mathcal{S}\llbracket\mathsf{P}\rrbracket\ (\mathsf{EXIT}\ \bot)) \\
\mathcal{S}\llbracket\textbf{skip}\rrbracket\ s & = & s \\
\mathcal{S}\llbracket x := \mathsf{E}\rrbracket\ s & = & \mathcal{E}\llbracket\mathsf{E}\rrbracket\ (x,s) \\
\mathcal{S}\llbracket\mathsf{S}\ ;\ \mathsf{T}\rrbracket\ u & = & \mathcal{S}\llbracket\mathsf{S}\rrbracket\ (\mathcal{S}\llbracket\mathsf{T}\rrbracket\ u) \\
\mathcal{S}\llbracket\textbf{return}\ \mathsf{E}\rrbracket\ s & = & \mathcal{E}\llbracket\mathsf{E}\rrbracket\ (x, \mathsf{RETURN}\ x\ s) \\
\mathcal{S}\llbracket\textbf{if}\ \mathsf{B}\ \textbf{then}\ \mathsf{S}\ \textbf{else}\ \mathsf{T}\ \textbf{fi}\rrbracket\ u & = & \mathcal{E}\llbracket\mathsf{B}\rrbracket\ (x, \mathit{if}\ \mathsf{VAR}\ x\ \mathit{then}\ \mathcal{S}\llbracket\mathsf{S}\rrbracket\ u\ \mathit{else}\ \mathcal{S}\llbracket\mathsf{T}\rrbracket\ u) \\
\mathcal{E}\llbracket\textbf{var}\ y\rrbracket\ (x,s) & = & x := \mathsf{VAR}\ y\ s \\
\mathcal{E}\llbracket\textbf{num}\ n\rrbracket\ (x,s) & = & x := \mathsf{VAL}\ n\ s \\
\mathcal{E}\llbracket\mathsf{E}\ â‘¡\ \mathsf{F}\rrbracket\ (x,s) & = & \mathcal{E}\llbracket\mathsf{E}\rrbracket\ (y, \mathcal{E}\llbracket\mathsf{F}\rrbracket\ (z, x := (y\ â‘¡\ z\ s))) \\
\mathcal{E}\llbracket\textbf{call}\ \mathsf{P}(\mathsf{E})\rrbracket\ (x,s) & = & \mathcal{E}\llbracket\mathsf{E}\rrbracket\ (y, x := \mathsf{CALL}\ (\mathcal{P}\llbracket\mathsf{P}\rrbracket, y)\ s) \\
\mathcal{P}\llbracket\textbf{proc}\ x\ \textbf{begin}\ s\ \textbf{end}\rrbracket & = & (\mathcal{S}\llbracket\mathsf{S}\rrbracket\ (\mathsf{EXIT}\ \bot), x)
\end{array}$$

The run-time operations indeed are very close to concrete machine instructions

$$\begin{array}{rcl}
\mathsf{EXIT}\ s\ \eta\ \delta & = & \delta\ \bot \\
\mathsf{RETURN}\ x\ s\ \eta\ \delta & = & \delta\ (\eta\ x) \\
x := \mathsf{VAR}\ y\ s\ \eta\ \delta & = & s\ \eta[x := \eta\ y]\ \delta \\
x := \mathsf{VAL}\ n\ s\ \eta\ \delta & = & s\ \eta[x := n]\ \delta \\
x := \mathsf{CALL}\ ((p,y),z)\ s\ \eta\ \delta & = & p\ \eta_0[y := \eta\ z]\ (\lambda v.s\ \eta[x := v]\ \delta) \\
x := y\ â‘¡\ z\ s\ \eta\ \delta & = & s\ \eta[x := \eta\ y\ â‘¡\ \eta\ z]\ \delta
\end{array}$$

In fact, the run-time operations are now so primitive that they can be implemented directly in some concrete machine code; the dump continuation would be realized as a stack of pairs of environments and return addresses. Another option would be using C as target code in which case primitive operations are inherited from C directly, in particular the procedure call and return. An actual compiler derived from our last interpreter generates the following code for the factorial example

```
fac(n){
     x10 = n;
     x9 = 0;
     x1 = x10 == x9;
     if(!x1){goto else;}
     x2 = 1;
     return (x2);
     goto fi;
  else:
     x5 = n;
     x8 = n;
     x7 = 1;
     x6 = x8 - x7;
     x4 = fac (x6);
     x3 = x5 * x4;
     return (x3);
  fi:
  exit (0);
};
```

9 Tail call elimination

Suppose the context condition holds that any variable appearing in a program is defined (occurs on the lhs of an assignment) before it is used (occurs on the rhs of an assignment), then the CALL instruction can be refined to pass the current environment η instead of the empty environment η_0.

$$x := \mathsf{CALL}\ ((p,y),z)\ s\ \eta\ \delta \quad = \quad p\ \eta[y := \eta\ z]\ (\lambda v.s\ \eta[x := v]\ \delta)$$

This modified CALL instruction allows tail recursive calls to be replaced by iteration. When a procedure returns by calling a procedure (either itself or another), that call is said to be a *tail call.* We can replace such a call by a jump, this not only saves time but also (stack) space.

$$\begin{aligned}
& x := \mathsf{CALL}\ ((p,y),z)\ (\mathsf{RETURN}\ x\ s)\ \eta\ \delta \\
= \quad & \text{evaluate CALL} \\
& p\ \eta[y := \eta\ z]\ (\lambda v.\mathsf{RETURN}\ x\ s\ \eta[x := v]\ \delta) \\
= \quad & \text{evaluate RETURN} \\
& p\ \eta[y := \eta\ z]\ \delta \\
= \quad & \text{definition VAR and GOTO} \\
& y := \mathsf{VAR}\ z\ (\mathsf{GOTO}\ p\ s)\ \eta\ \delta
\end{aligned}$$

Incorporating this peephole optimization into our translation scheme gives

$$\mathcal{S}[\![\textbf{return}\ (\textbf{call}\ \mathsf{P}(\mathsf{E}))]\!]\ s \quad = \quad \mathcal{E}[\![\mathsf{E}]\!]\ (x, \mathsf{GOTO}\ p\ s)\ \textit{where}\ (p,x) = \mathcal{P}[\![\mathsf{P}]\!]$$

Note that this definition is not homomorphic, but could be made so easily by letting the parser detect it and introducing an additional constructor in the abstract syntax.

10 Conclusions and future work

We hope to have convinced the reader that a not totally unrealistic implementation for a simple imperative language can be derived from its denotational semantics in a few (mechanical) transformation steps. The crucial thing that makes the transformation of an inefficient into an efficient interpreter possible is that there is a homomorphic embedding of the abstract into the concrete semantic domain. For first-order languages like *C* this is easy, but things get notoriously difficult when the semantic domains become more complicated. Most troublesome in this respect is the contravariance of the function space functor. At present we lack nice calculation rules for recursive domains with function spaces. This effectively means that our techniques does not extend directly to interpreters for functional languages. Historically our work on compiler derivation started by the desire to unify the wealth of abstract machines for implementing functional languages by calculating each one of them from the generic meta-interpreter for the lambda-calculus (The seminal work of Lester [16] on the G-machine is still unique in this respect). We have good hopes however that the induction principles for recursively defined domains of Pitts [25] are the right theoretical tool to tackle this problem.

Besides the need for a stronger theory, practical application of the techniques for deriving compilers as proposed in this paper requires machine assistance for sure. First of all an automatic free theorem generator is indispensable, it is rather tedious to figure out fusion laws like that for Dump by hand. We have written a simple such a program, but is still requires too much manual simplicifation to make the resulting theorems palatable. Secondly it would be nice to have an intelligent editor that could handle all the dump transformation steps that appear in the calculations. We don't know whether one wants to automate all calculations (even if possible) as you want a thight control on the new run-time operations being synthesized. All interpreters and compilers described in this paper have been implemented in Gofer (NOTE: perhaps it would be interesting to include the Gofer code as an appendix). Apart from the use of a monad to provide an explicit source of fresh variables there is no difference between the interpreters written in Gofer and those presented in the paper. When factoring an actual compiler from an interpreter some care has to be taken to produce the correct (finite) textual representation of the (recursive) target program, i.e. we have to generate a string that when parsed yields the proper abstract syntax tree of the target language. This aspect is not properly expressed in the factorization theorem, but did not turn out to be a real problem though it could make the *automatic* factorization of a compiler from an interpreter non-trivial.

acknowledgements

Graham Hutton and Johan Jeuring both provided valuable comments on this paper.

References

[1] Aho, Sethi, and Ullman. *Compilers, Techniques and Tools.* Addison-Wesley, 1986.

[2] Richard Bird. Constructive functional programming. In M. Broy, editor, *Marktoberdorf International Summer school on Constructive Methods in Computer Science*, NATO Advanced Science Institute Series. Springer Verlag, 1989.

[3] R.M. Burstall and P.J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4, 1969.

[4] L.M. Chirica. *Contributions to Compiler Correctness.* PhD thesis, University of California at LA, USA, 1976.

[5] Peter Dybjer. Using domain algebras to prove the correctness of a compiler. In *LNCS 182.* Springer Verlag.

[6] Maarten Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer. Translating attribute grammars into catamorphisms. *The Squiggolist*, 2(1), 1991.

[7] Maarten Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, dept of Informatics, Enschede, The Netherlands, 1992.

[8] Peter Freyd. Recursive types. July 1989.

[9] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24(1):68–95, 1977.

[30] J.W. Tatcher, E.C Wagner, and J.B Wright. In Neil D. Jones, editor, *LNCS 94: Workshop on Semantics Directed Compiler Generation*. Springer, 1980.

[31] Phil Wadler. Theorems for free ! In *Proc. FPCA'89*, pages 347–359, 1989.

[32] Philip Wadler. The essence of functional programming. In *POPL19*, 1992.

[33] D.A. Watt. Executable semantic descriptions. *Software Practice and Experience*, 16(1):13–43, 1986.